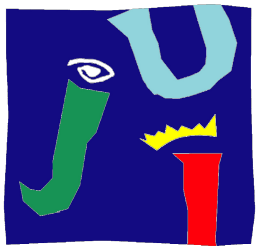
Procedural generation of optimized maps for survival video games

End-of-Degree project

By

JESÚS BACHILLER CABAL



UNIVERSITAT JAUME·I

Degree of Design and Development of Video games
UNIVERSITY JAUME I

Supervisor: DR. RAÚL MONTOLIU COLÁS.

JUNE 3, 2018

Abstract

The goal of this End-of-Degree project has been to create a random world that adapts to the needs of the developed video game in this project. In the last years, there has been a great stake for random generation techniques in the video game industry. These techniques make it possible for players to feel a different and unpredictable game every time they play.

The Unity3D graphics engine has been used for the development of the project. This work has two distinct parts: the procedural generation of the optimized terrain for video games (main theme of the project) and a survival video game where the result of the first part is shown. On the one hand, the generation has been divided into two parts: generation of a mountainous terrain and the generation of game resources throughout the map. On the other hand, the typical video game development process has been followed: writing of the Game Design Document, modeling of the objects and resources of the game and character programming, interactions, mechanics, etc.

Finally, a basic and optimized game with a procedurally generated terrain and with the indicated characteristics in Game Design Document has been obtained.

Keywords: Survival video game, Unity3D, terrain, procedural generator, optimization.

TABLE OF CONTENTS

List of Tables

List of Figures

Figure **Page**

CHAPTER 1

TECHNICAL PROPOSAL

This chapter constitutes the Technical Proposal of the End-of-Degree project at the Degree of Design and Development of Videogames of the Jaume I University. This work will consist of the development of a survival video game in Unity3D [23] where the player will have to collect and use resources that find around the map to survive and advance the game. Player will also have to combine them to generate new resources, tools or weapons. Its distinctive feature is the use of algorithms in order to procedural generation of a playable and optimized map. They will allow adapting the generated terrain to the needs of each player and his computer.

1.1 Introduction and motivation

In recent years, the procedural techniques have experienced an overwhelming growth in the video game industry, in large part thanks to video games that have explored and exploited this field as Minecraft [15] or No Man's Sky [16].

Thanks to these techniques it is possible for the player to feel new and different experiences every time he/she starts a game since they can be used to create whole random worlds. The primary motivation of this project is to develop a survival video game where the procedural generation of the map is an essential part of the complete game development.

In the game, the player will play the role of the main character who is on a vast island (or set of islands, it will depend on the procedural generation of the game terrain) and without communication with the rest of the world. The player will have to survive with the resources

that will have to find and will have to create, combining some resources with others, to build a machine that allows the player to get out of there.

The game will be a first-person 3D game where the Lowpoly technique and a pastel colour palette will be used for the artistic section.

For the procedural generation of the game map, Perlin Noise function will be taken as the starting point. From here, the different possibilities that exist will be studied. Finally, the most appropriate technique will be implemented to achieve the objectives of this project (see Section 1.4).

1.2 Subjects related

The main subjects related to this project are (sorted by university code):

- VJ1205 - English
- VJ1207 - Programming II
- VJ1212 - Graphic expression
- VJ1221 - Graphic computing
- VJ1224 - Software engineering
- VJ1227 - Game engines

1.3 Tools

The tools used in this project are:

- **Unity3D** for the game engine.
- **Blender** to model and animate.
- **Visual Studio with C#** to scripting.
- **Photoshop** to make sketches, figures and textures.
- **Overleaf** to write end-of-grade work memory.

1.4 Goals

1. Creation of the base of a video game of survival so that, later, it can continue to improve aspects of the game and adding new ones.

2. Achieve a physical and logical realism in the procedurally generated terrain.

3. Obtain a logical coherence in the procedural generation of resources by the map.

4. Get an optimized game that can be played by any player on any standard computer.

5. End up having an attractive visual section for the public. This includes: models, textures, lighting, animations, etc.

1.5 Planning

Task	Hours		Task	Hours		Task	Hours
T0: Technical Proposal.	10		T4: Programming of main character movement.	10		T6: Programming of procedural the generation of resources.	30
T0: Analysis and design document.	20		T5: Search for information on the procedural generation of terrain in games.	15		T7: Programming of interaction of the character with the environment.	15
T1: Modeling and texturing of the main character.	15		T5: Programming of the procedural generation of the terrain.	30		T7: Programming of collection and resource crafting mechanics.	20
T2: Modeling and texturing of game resources.	15		T5: Search of information of optimization of terrain in games.	10		T8: Memory.	50
T3: Modeling and texturing of game structures.	20		T5: Programming of Optimized procedural terrain.	20		T9: Preparation of the final presentation.	20
						Total hours	**300**

TABLE 1.1. An initial approximation of project tasks and estimated hours.

1.6 Expected results

The expected global result is the creation of the basis of a survival video game where the player can interact with the environment and its resources. The pursued result is to generate an optimized procedural terrain and resources that constitute the world of the game of survival.

CHAPTER 2

GAME DESIGN DOCUMENT

In this chapter, the Game Design Document (GDD) of the developed game to show in action the main objective of this End-of-Degree Project will be carried out.

2.1 Introduction

2.1.1 Game name

ALONE

2.1.2 Genre

It is a 3D survival game in an open world.

2.1.3 Purpose and target audience

Develop a video game that shows the potential of the procedural generation in open-world games.

Alone is aimed at casual players of all ages looking for a quiet, leisurely and free game. Players must have time to enjoy the game and learn about the game.

2.1.4 Game concept

Alone is a first-person survival game where the protagonist will be in a geographical area utterly unknown to him. He/she seems to be alone since there is no indication of human presence there.

Without any help, he/she will have to survive, feed and progress to find a way out of that place. As time passes and the protagonist goes exploring, he realises that he is isolated in an island or set of islands.

The protagonist will have to go collecting natural resources, such as wood or minerals. With these resources, he will have to build tools, structures and other resources. These will allow him to progress until he achieve out of that place.

2.2 Gameplay

2.2.1 Game mechanics

The player will have the ability to craft. As defined by the Gamerdic website, the craft is described as —Making objects from existing ones or from essential elements that can be collected in a game.— [4]. The player will find resources throughout the map that can not be used to advance the main objective of the game. For this reason, crafting is the main mechanics of the game. There will be objects that the player will cannot directly craft. For this type of objects, special structures will be needed.

Another mechanic is the exploration. Being an open world where the player has freedom of movement, it is essential to find the natural resources necessary to advance in the game, since these resources are located in specific geographical areas of the map (see Section 2.2.6).

Feeding is another challenge that the player has to face to survive. For this reason, eating is an essential game mechanic. The player can eat natural food (for example seeds) or toasted food (for instance cooked seeds). Depending on the type of food that he will ingest, the hunger of the character will more or less calm down.

Besides, to calm hunger and to avoid death, eating makes the character recover life. Just as it happens to satisfy hunger, depending on the food taken by the player, he will improve more or less life.

2.2.2 Goals of the game

The main goal of Alone is to build a sturdy boat that allows the player to leave the island as quickly as possible. For this, he will have to discover new objects and structures that he can create from resources that he has and finds. So another goal of the game is to know and learn the possible combinations of objects.

2.2.3 Controls

The game is controlled with a keyboard and a mouse. The player will use keys **A**, **W**, **S** or **D** to move the character to the left, forward, back or right, respectively. He/she also uses mouse drag to guide the vision of the game and key **SPACE** to jump. The **LEFT** button mouse will be used to interact with any element of the game. And to open the inventory of the character, the player will use the key **E**. These controls can be seen, more briefly, in Table 2.1.

Button / Key	Actión
A	left move
D	right move
W	forward move
S	backward move
E	open inventory
SPACE	jump
drag mouse	move the camera
left button mouse	interact with elements

TABLE 2.1. Game controls

2.2.4 Game character

2.2.5 World

As has been mentioned earlier in the Game concept (see Section 2.1.4), Alone is located in a geographical area isolated from the rest of the world. The map of the game is an island or set of islands where the player can find different geographic areas with common characteristics. These areas are known as Biomes, and there are six different types:

- **Meadow**: Biome that is characterized mainly by being located on flat land and being full of vegetation where there are a lot of herbs and plants and where there are few trees. In addition to vegetation, here are the Coal Mines of the island. It is a biome where light colours.

- **Forest**: Biome also located on flat land. It is a biome where trees abound, and dark brown and dark green colours predominate. It is a place where the level of luminosity is low because the trees avoid sunlight from entering the forest.

- **Beach**: This biome is located on the coast and is one of the biomes where the player can find Iron Mines. It is a biome where sand predominates, although rocks and stones can also be found. It is a very lighting biome.

- **Low Mountain**: This Biome is located at the base of the mountains. It is a place where vegetation is scarce but varied. The mineral that the player can be found here is the coal. It presents colors similar to those that have forest but more illuminated because there isn't so much concentration of trees.

- **Half Mountain**: Biome that is in the middle heights of the mountains. It is characterized by being a rocky terrain and without vegetation. These areas are one of the main sources of iron.

- **High Mountain**: It is a continuation of the Half Mountain Biome. It is also a rocky biome with sparse vegetation. But being at high altitudes, it is covered with a layer of snow.

2.2.6 Resources and crafts

In this section, all the objects with which the player can interact are described. They are divided into two types of resources: On the one hand, natural resources are those that are found in nature and that have not been modified by any human; On the other hand, the artificial resources are derived from natural resources.

In the resource relationship diagram (see Figure 2.1), all the objects of the game and their relationship to each other are represented. For example, at the top left is the tree and to its right, the trunk joined with an arrow towards the trunk. This means that the trunk is obtained from the tree. Resources that do not come from any other resources are the natural resources.

The player can craft some resources by himself/herself. But there are other resources that can be only created in special structures such as tables or furnaces.

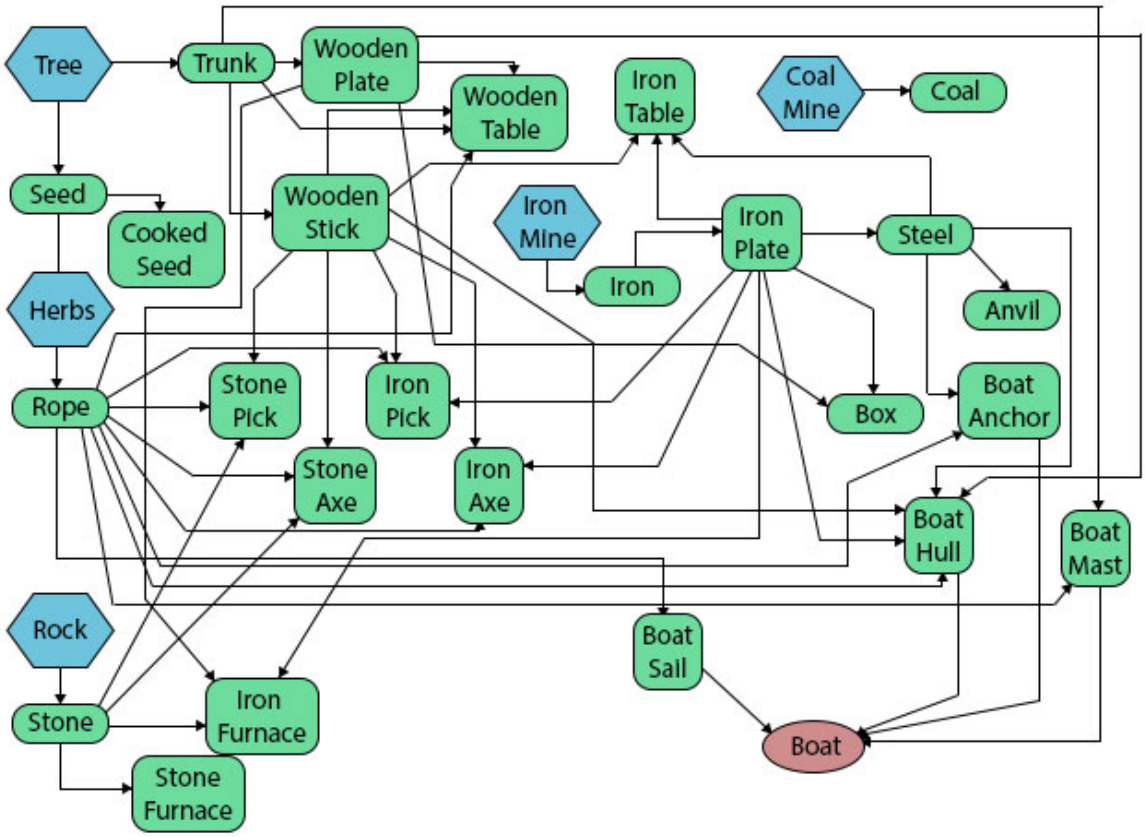


FIGURE 2.1. Resource relationship diagram. Each arrow indicates what can be obtained from each resource. The resources inside the blue hexagons are natural resources.

2.2.6.1 Natural resources

A list of all natural resources of the game along with a short description is shown:

Name	Description
Rock	Rocky concentration very common in the Beach biome and High and Half Mountain biomes. The inside does not contain any other type of material. From it, Stones can be extracted.

Follow on the next page.

Name	Description
Mine	They are rocks that are formed by stones and another additional mineral. 2 types of mines can be found: • Iron Mine: It contains Iron Ore. This type of mine is common in the Half Mountain and Beach biomes. • Coal Mine: It contains Coal Ore. This type of mine is common in the Low Mountain and Meadow biomes.
Tree	It can be found throughout the island sporadically except in the Half and High Mountain. Although, it is very abundant in the Forest biome. The wood, an essential resource for survival on the island, can be obtained from it.
Herbs and Plants	Like trees, It can be found throughout the island sporadically except in the Half and High Mountain, but it abounds in Meadow Biome.

TABLE 2.2. Natural resources and descriptions.

2.2.6.2 Artificial resource

A list of all artificial resources of the game along with a short description is shown:

Name	Description
Trunk	It is obtained from the trees.
Stone	It is obtained from rocks and mines and is useful for the construction of the first tools and for the construction of certain structures.
Wooden plate	It is obtained from the trunks.
Wooden stick	It is obtained from the trunks.
Iron	It is obtained from the iron mines.
Iron plate	It is obtained from the iron. A furnace is needed to obtain it.
Rope	It is obtained from the herbs and plants.
Seed	It is obtained from the herbs, plants and trees. It serves to feed. It calms little the hunger of the player.
Cooked seed	It is obtained from the seeds and wooden plates. A Furnace is needed to obtain it. It serves to feed. It calms a lot the hunger of the player.
Wooden table	It is obtained from the wooden plates, wooden sticks, trunks and ropes. It serves to make the crafts that cannot be done by himself/herself.

Follow on the next page.

Name	Description
Iron table	It is obtained from the iron plates, wooden sticks, steel and ropes. It serves to make the crafts that cannot be done in the wooden table.
Stone pick	It is obtained from the stones, wooden sticks and ropes. It can only be built in any table (Wooden or iron table). It serves to be able to mine minerals more quickly.
Stone axe	It is obtained from the stones, wooden sticks and ropes. It can only be built in any table (Wooden or iron table). It serves to be able to chop down trees more quickly.
Iron pick	It is obtained from the iron plates, wooden sticks and ropes. It can only be built in a iron table. It serves to be able to mine minerals more quickly.
Iron axe	It is obtained from the iron plates, wooden sticks and ropes. It can only be built in a iron table. It serves to be able to chop down trees more quickly.
Coal	It is obtained from the coal mines.
Stone furnace	It is obtained from the stones. It can only be built in any table (Wooden or iron table). It serves to fuse minerals and to cook food. It needs coal to operate.
Iron furnace	It is obtained from the iron plates, wooden plates and wooden sticks. It serves to fuse minerals and to cook food efficiently. It needs coal to operate.
Box	It is obtained from the iron plates and wooden plates. It can only be built in any table (Wooden or iron table). It serves to store resources.
Steel	It is obtained from the iron plates. An iron furnace is needed to obtain it.
Anvil	It is obtained from the steel, wooden plates and ropes. An iron table is needed to obtain it.
Boat hull	It is obtained from the steel, wooden plates, iron plates, wooden sticks and ropes. An iron table is needed to obtain it.
Boat anchor	It is obtained from the steel and ropes. An anvil is needed to obtain it.
Boat mast	It is obtained from the trunks, ropes and wooden sticks. An iron table is needed to obtain it.
Boat sail	It is obtained from the ropes. An iron table is needed to obtain it.
Boat	It is obtained from the boat hull, boat anchor, boat mast and boat sail. An iron table is needed to obtain it. It is the goal of the game.

TABLE 2.3. Artificial resources and descriptions.

2.3 Flow diagram

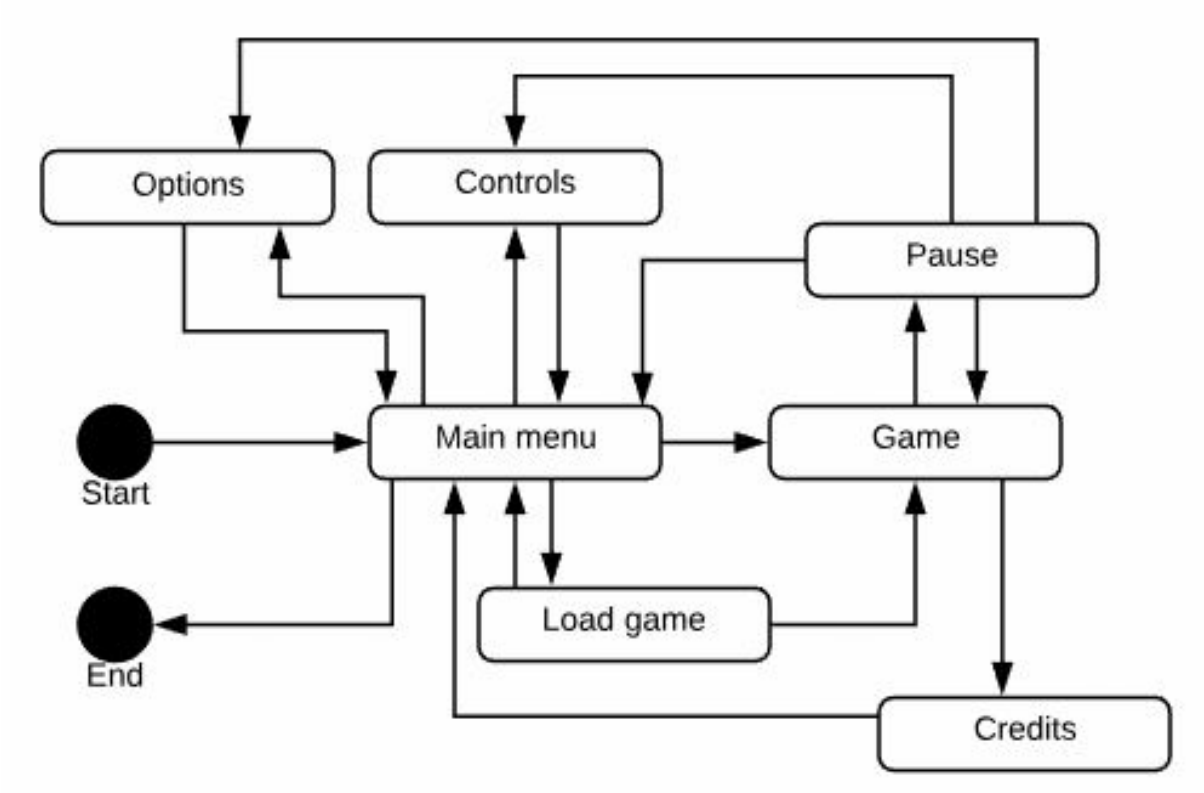


FIGURE 2.2. Flow diagram of the game. It obtained from the Lucidchart website [14].

The first screen will be "Main menu" whenever the game starts. From here, the player can go to the screens: "Controls" (to show off the game controls), "Options" (to modify graphics and sound options of the game), "Load game" (to load a saved game) and "Game" (to start a new game). Back to the menu is possible from all these screens, except the Game screen. To return from it, the game has to be paused. When the game ends ("Game" screen) it will automatically direct to the "Credits" and then to the "Main menu". It is only possible to close the game (the application) from the "Main menu". So if the player is in a game and wants to close the application, he will have to go to the "Main menu" first.

2.4 User interface

2.4.1 Main menu

The main menu will have five basic buttons:

- **New game**: It starts a game from the beginning.
- **Load game**: It shows the games that the player has saved.
- **Exit**: Close the application.
- **Options**: It allows to see and change the graphic and sound options.
- **Controls**: It shows the game controls.

2.4.2 In-game HUD

The HUD of the game is designed to have the information that a real person in that situation. The screen will display only the level of hunger and life of the character and the objects that he has ar that moment. Likewise, the HUD will not have a mini-map since this information is not available in a real situation.

2.5 Art

The game objects will be three-dimensional models and will be done with the Low Poly technique to emphasize the simplicity of the style. Likewise, it will also help the player to focus on the game instead of on the elements. These models will be formed by simple textures with a single pastel colour.

The following sections show sketches of the game objects and some visual references.

2.5.1 Sketches

In this section, some sketches of the game resources are shown. Those sketches will serve to make the 3-D models (see Figure 2.3). More sketches can be seen in the following link: Sketches.

2.5.2 References

A video game that can be taken as a visual and gameplay reference is Astroneer. This game is an Early-Access Indie Space Themed Exploration & Survival-Crafting Games developed by System Era [2]. The Figure 2.4 shows more example how the game could be seen (Low Poly technique).

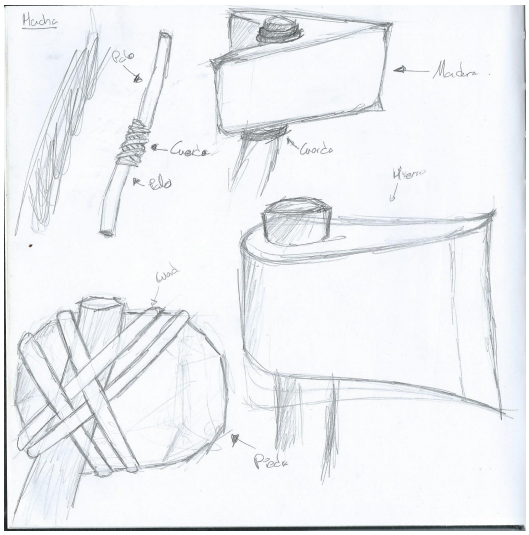


FIGURE 2.3. Game sketches.

FIGURE 2.4. Visual references of the game.

CHAPTER 3

PROJECT DEVELOPMENT

This chapter explains all the tasks done in the project are developed and explained. Especially this section is focused on the issue of procedural generation since it has a lot of weight in this project.

3.1 Task 0: Technical Proposal and Analysis and design document

This task consists of writing the two documents that make up the chapters one and two of this memory. The technical proposal specifies what is to want to be carried out in this project, as well as small planning, objectives, and tasks. The GDD exposes more concrete data about the game that is going to be implemented.

Overleaf [18] has been used to write these documents (see Figure 3.1). It is a free service that lets you create, edit and share your scientific ideas easily online using LaTeX, a comprehensive and powerful tool for scientific writing [17].

3.2 Task 1, 2 and 3: Modeling, texturing and animation of: main character, game resources and game structures

Blender has been used to carry out these three tasks. It is the free and open source 3D creation suite. It supports the entirety of the 3D pipeline: modeling, rigging, animation, simulation, ren-

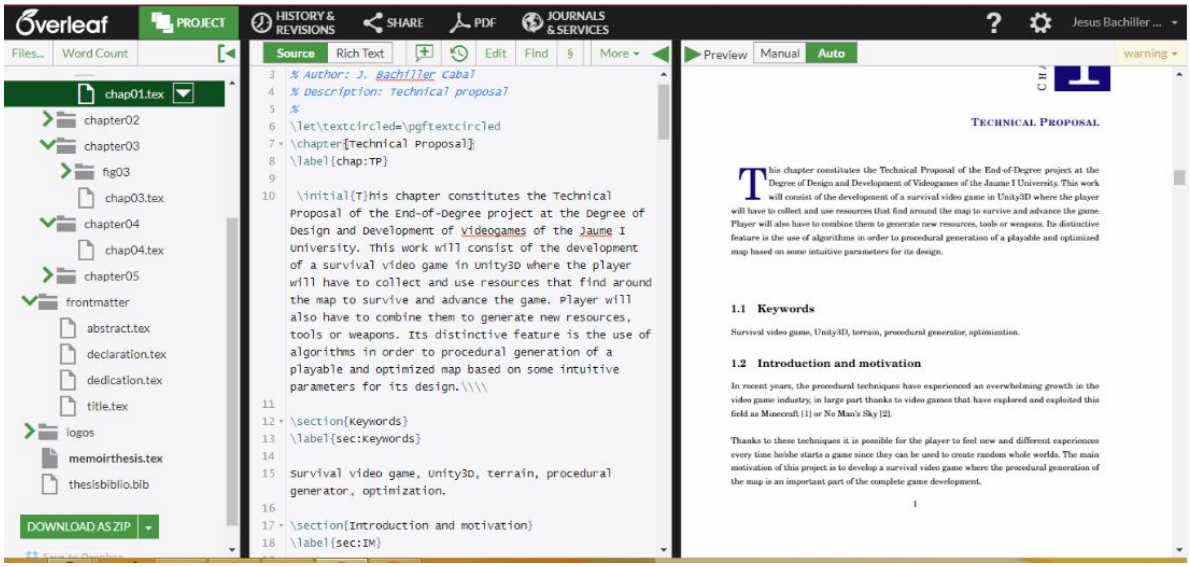


FIGURE 3.1. Sceen of Overleaf.

dering, compositing and motion tracking, even video editing and game creation [7].

For modeling, the Lowpoly technique has been used with textures of a single color as already explained in the GDD of the game. Making use of this technique has been decided to prevent a possible low performance of the game since the size of the terrain is expected to be large with a lot of resources.

The resources have been modeled following the sketches proposed in the GDD. However, the main character has only been represented with two polygons. It has allowed us to use the time for modeling in other tasks of the project.

3.3 Task 4: Programming of main character movement

The programming of the player's movement has not had to be programmed. For this project, a Standard Asset Packages of Unity has been used. They are Asset collections pre-made and supplied with Unity [21].

Only the right package has been imported and has been adjusted the necessary values to get an adequate movement to the game.

3.4 Task 5: Procedural generation of optimizated terrains

3.4.1 Noise algorithms

The noise algorithms are those that are responsible for generating random or pseudorandom waves from some input data. So for each data input, a different n-dimensional noise wave is generated. These algorithms are very useful to model natural qualities such as clouds, landscapes or marble textures (see Figure 3.2).

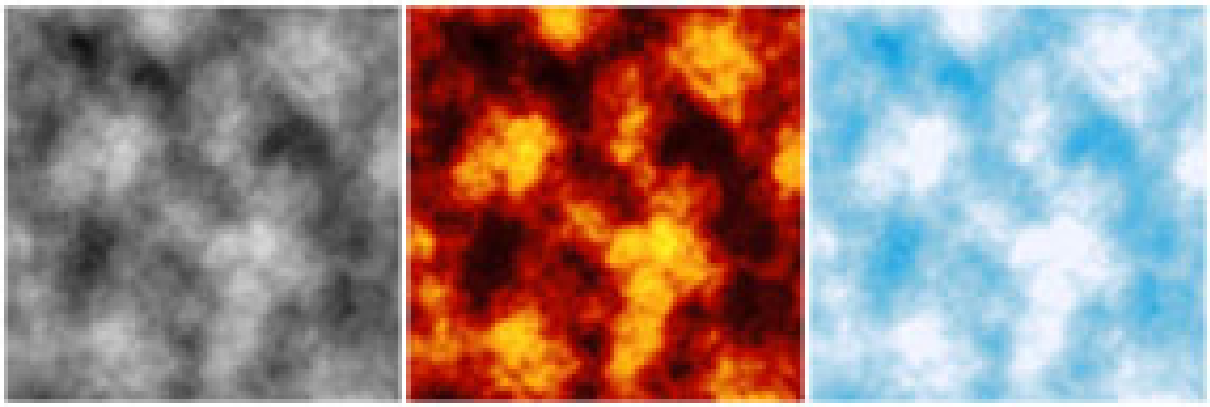


FIGURE 3.2. Natural qualities modeled with the Perlin Noise function. On the left side, a grayscale noise texture. In the center, the texture used to model fire. And on the right side, used to model clouds.

There are different noise algorithms: Simplex Noise algorithm or Diamond-Square algorithms. However, Perlin Noise will be used in this project. The main reason is that Unity provides an implementation of that algorithm in its Mathf class [6]. It will speed up the development of the project since it is based on the adaptation of a noise algorithm for the generation of terrain, and is not based on the implementation of the said algorithm. Another reason is that there is a lot of information about the procedural generation with Perlin Noise as an instance [10] and [19].

3.4.2 Perlin Noise

Perlin Noise is an algorithm developed by Ken Perlin in 1982 for the Disney film Tron [11]. It is a gradient-based algorithm that generates a coherent noise; this means that there is little variation between two adjacent points of the noise wave. Therefore, this algorithm is ideal for creating of continuous and smooth 3D terrains.

It is inevitable to mention some terms when working with waves: amplitude and frequency. On the one hand the amplitude at a point of the wave refers to the distance between that point and Y-axis; On the other hand, the frequency is the number of cycles per unit length along the Y-axis, this refers to the X-axis (see Figure 3.3).

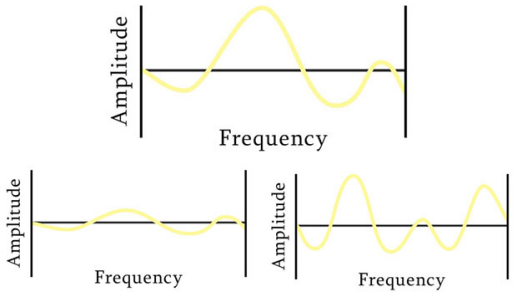


FIGURE 3.3. Relation between frequency and amplitude. Same wave with different amplitudes and frequencies.

If a section of the Perlin Noise wave is taken, something like a mountainous terrain could be obtained but not a natural terrain because its curve is still too soft (for example the upper wave of the Figure 3.3). For this reason, it is necessary to add detail but keeping its general form. It is achieved by generating multiple levels of noise layers, commonly called Octaves.

It is wanted that the detail of the curve is increased in each octave: It will be done in a 2:1 ratio, that is, multiplying the frequency by two and dividing the amplitude also by two. It is possible to change this ratio, and it would not affect the final result, it is simply a personal decision. To control this ratio two terms are defined: Lacunarity y Persistance. On the one hand, Lacunarity is responsible for controlling how the frequency changes in each octave (multiplying by two), and on the other hand Persistence controls the influence of each octave in the final wave, that is, it controls the amplitude (dividing by two). See Figure 3.4.

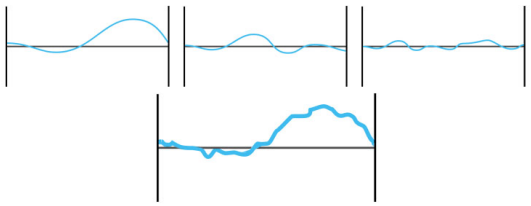


FIGURE 3.4. The three upper waves represent three different octaves of the same wave with a 2:1 ratio. The lower wave is the result of the sum of the three octaves.

3.4.3 Procedural terrain generation

3.4.3.1 2D noise map

The generation of the terrain begins by obtaining the 2D noise map. To do this, a method that generates this map is required. In this method, the values of Lacunarity, Persistence, and Octaves are needed. The Seed attribute has also been added to generate random numbers controlled by said attribute. So thanks to this attribute, the same noise map will be obtained if the same value is inserted.

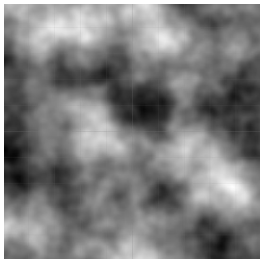


FIGURE 3.5. Display of the noise map (grayscale) in a 2D texture in Unity.

As it can be observed in Figure 3.6, the method starts by getting the displacement values for each noise map octaves. Subsequently, each noise map value, still empty, is updated with the noise value from the Perlin Noise function [6] in each octave level. Also in this process, the Amplitude and Frequency values are updated as explained in Section 3.4.2. Finally the noise map is normalized to obtain a map in a range between zero and one: visually the map will display as a grayscale (see Figure 3.5).

Once the function generates the noise map, it can be displayed in grayscale in Unity. To do this, the values of the noise map are interpolated. The zero is the black color, the one is the white color and the rest of values are grays. Finally, these colors are assigned to each pixel of texture from Unity object (see Figure 3.7).

Now, rather than interpolating the noise map value, ranges of values between zero and one are set and a color is assigned to each field, the first 2D images, that may be a game map, can already start to be observed (see Figure 3.8).

The result of this section can be seen in a video by clicking here.

```
float[,] generateNoiseMap(size, seed, octaves, persistance, lacunarity) {
  map = 2D-array of floats with width "size" and height "size";

  octavesOffsets = array of 2D points with length equal to octaves;
  for(i from 0 to octaves) {
    octavesOffsets[i] = 2D-point(getRandomFromSeed(seed), getRandomFromSeed(seed));
  }

  for (y from 0 to size) {
    for(x from 0 to size) {
      amplitude = 1; frequency = 1; noise = 0;

      for (i from 0 to octaves) {
        xValue = (x  + octavesOffsets[i].x) * frequency;
        yValue = (y + octavesOffsets[i].y) * frequency;

        float perlinValue = PerlinNoise(xValue, yValue) * 2 - 1;
        noise += perlinValue * amplitude;
        amplitude *= persistance; frequency *= lacunarity;
      }

      map[x, y] = noise;
    }
  }

  Normalize(map);
  return map;
}
```

FIGURE 3.6. Pseudocode to generate the 2D noise map from the Size, Seed, Octaves, Persistance and Lacunarity values.

```
public void DrawNoiseMap() {
  map = GenerateNoiseMap(100, 0, 8, 0.5, 2);
  texture = setColorOfPixels(Interpolate(from black to white map values));
  Texture of Unity plane object = texture;
}
```

FIGURE 3.7. Pseudocode to display the noise map in Unity (grayscale).

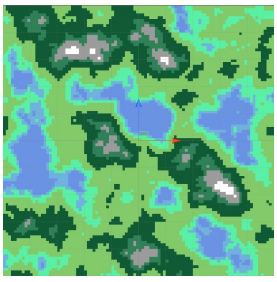


FIGURE 3.8. Noise map with colors. Result of running the DrawNoiseMap() function (Figure 3.7), adding eight ranges of values.

3.4.3.2 3D mesh

The next step is to generate the 3D mesh from the noise map (obtained in Section 3.4.3.1). First, a plane with the desired size will be generated, and subsequently, the height of each vertex will be established according to the noise map.

There are different representations of polygonal meshes: independent faces, shared vertices, strips, etc. In this case, independent faces representation will be used. The main reason is the type of shading that this project wants to achieve: Flat Shadow, that is, each face has a unique color. Unity calculates the illumination of the triangles from the normals of his three vertices. Therefore, two vertices in the same position but from two different triangles have to be stored independently because each vertex will have a different normal (see Figure 3.9).

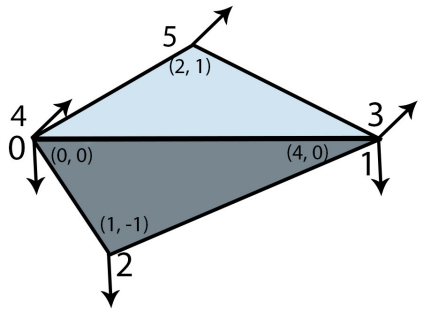


FIGURE 3.9. Mesh with independent face representation. The numbers represent the vertices, and the arrows represent the normals.

On the one hand Unity stores a 3D point list, which represents the mesh vertices; On the other hand, it stores an integers list, which represents the edges of the mesh triangles. It also saves different types of data such as the normal of each vertex or the texture coordinates (UV's). For example in the Figure 3.9, a 2 x 2 mesh will store six vertices and six edges. If it is generalized, a w x h mesh will store:

- Vertices = number of squares * triangles per square * vertices per triangle = [(w-1) * (h-1)] * 2 * 3 = [(2-1) * (2-1)] * 6 = [1 * 1] * 6 = 6.
- Edges = same as vertices.

The algorithm developed to generate the mesh with independent faces has been divided into two parts. First, a continuous mesh is generated, that is, with shared vertices; later it is divided into separate triangles (independent face representation).

For the first part of the algorithm, the vertices and edges arrays are generated. The edge array will have the same size as explained above. However, the size of the vertex list will be smaller because the mesh will have shared vertices: specifically, the size is w * h (the size will be four for the example of the Figure 3.10).

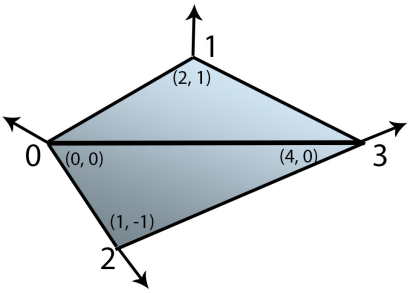


FIGURE 3.10. Mesh with shared vertices. The numbers represent the vertices, and the arrows represent the normals.

```
public mesh CreateMesh(heightMap) {
  w = width of heightMap; h = height of heightMap;
  mesh = new mesh with width w and height h;
  size of vertexList = w * h;
  size of edgeList = (w-1) * (h-1) * 6;
  index = 0;

  for(y from 0 to h) {
    for (x from 0 to w) {
      vertexList[index] = 3D-point(x, heightMap[x, y], y);

      if(x < w - 1 and y < h - 1) {
        edgeList.AddTriangle(index, index + w + 1, index + w);
        edgeList.AddTriangle(index + w + 1, index, index + 1);
      }

    index++;
    }
  }
  vertex list of mesh = vertexList;
  edge list of mesh = edgeList;

  mesh = FlatShading(mesh);
  calculate normals of mesh;

  return mesh;
}
```

FIGURE 3.11. First part of the algorithm to generate a 3D mesh.

The working of the first part of the algorithm can be seen in Figure 3.11. When a new mesh is creating, the vertex and edge lists are initialized. The size used is to store shared vertices. Then the vertices and edges are added. To store the edges in the list, the indices of the vertices that make up the triangles are saved as follows:

- Triangle 1 is composed of: (i, i + w + 1, i + w).
- Triangle 2 is composed of: (i + w + 1, i, i + 1).

So the edge list of the vertex with index 0 is: (0, 3, 2, 3, 0, 1).

Note the use of the if sentence to avoid accessing the last row and the last column. An undesired mesh and execution errors would be generated if this restriction were not made because when a vertex is evaluated, the two triangles that are below and to the right are added.

After executing the first part of the algorithm in Figure 3.10, the following lists would be available:

- Vertex list: [(0,0), (2,1), (1,-1), (4,0)].
- Edge list: [0, 3, 2, 3, 0, 1].

```
public mesh FlatShading(mesh) {
  length of vertexList_IndepFaces = Length of edgeList of mesh (With shared vertices);
  length of edgeList_IndepFaces = Length of edgeList of mesh (With shared vertices);

  for (i from 0 to Length of edgeList of mesh) {
    edges = edgeList of mesh;
    vertices = vertexList of mesh;

    vertexList_IndepFaces[i] = vertices[edge[i]];
    edgeList_IndepFaces[i] = i;
  }

  vertexList of mesh = vertexList_IndepFaces;
  edgeList of mesh = edgeList_IndepFaces;

  return mesh;
}
```

FIGURE 3.12. Second part of the algorithm to generate a 3D mesh.

The second step of the algorithm (see Figure 3.12) consist of to transform the mesh to independent faces. To do this, the edge list of the shared vertices mesh is traversed to know the triangles of this mesh (the result of the first step), and later the vertices of these triangles are stored

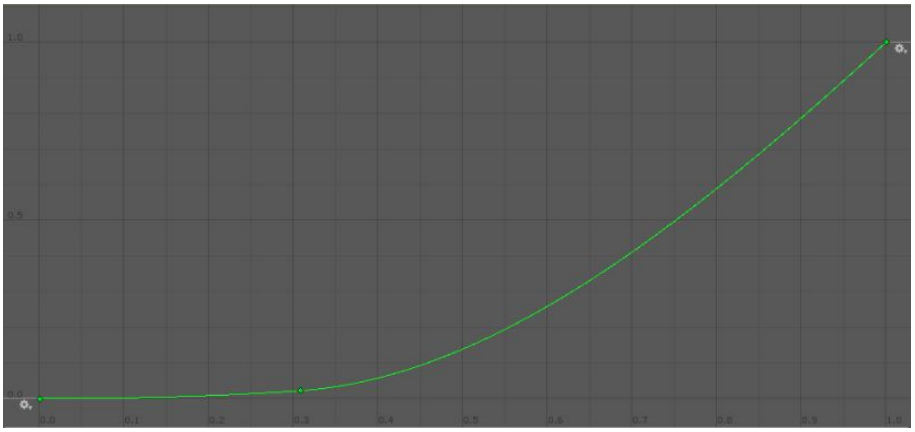


FIGURE 3.13. Curve to be able to specify how much and when the multiplier affects the mesh.

independently in a new vertex list (with the same size as the edge list). After executing the second part of the algorithm in Figure 3.9, the following lists would be available:

- Vertex list: [(0,0), (4,0), (1,-1), (4,0), (0,0), (2,1)]
- Edge list: [0, 1, 2, 3, 4, 5]

To complete this algorithm, a function that is already implemented in Unity is used: RecalculateNormals(). This function calculates the normal of each vertex as the sum of the normals of each triangle to which the vertex belongs. Each vertex will have the same normal as the triangle to which it belongs since it only pertains to the said triangle. That is, the normal of each vertex will be the cross product of the vectors composed of the three vertices of its triangle.

However, if this algorithm is executed, the resulting mesh is too flat since the values of the noise map are in a range between zero and one. For this reason, the algorithm has been completed with: A multiplier to accentuate the height and a curve to be able to specify how much and when the multiplier affects the mesh (see Figure 3.13). Thank the combination of the multiplier and the curve is possible to accentuate the mountainous areas.

For the curve in Figure 3.13, the multiplier barely affects the noise map values that are below 0.3. And for the rest of values, the multiplier affects more the greater the noise map value. For example, for a noise value equal to 0.75 and the multiplier equal to 25, the final value will be equal to 9.37 (25 * 0.5 * 0.75); and for a noise value equal to 0.55, final value will be equal to 2.75 (25 * 0.2 * 0.55).

The result of executing this algorithm in Unity can be seen in Figure 3.14 and it can be seen in a video by clicking here.

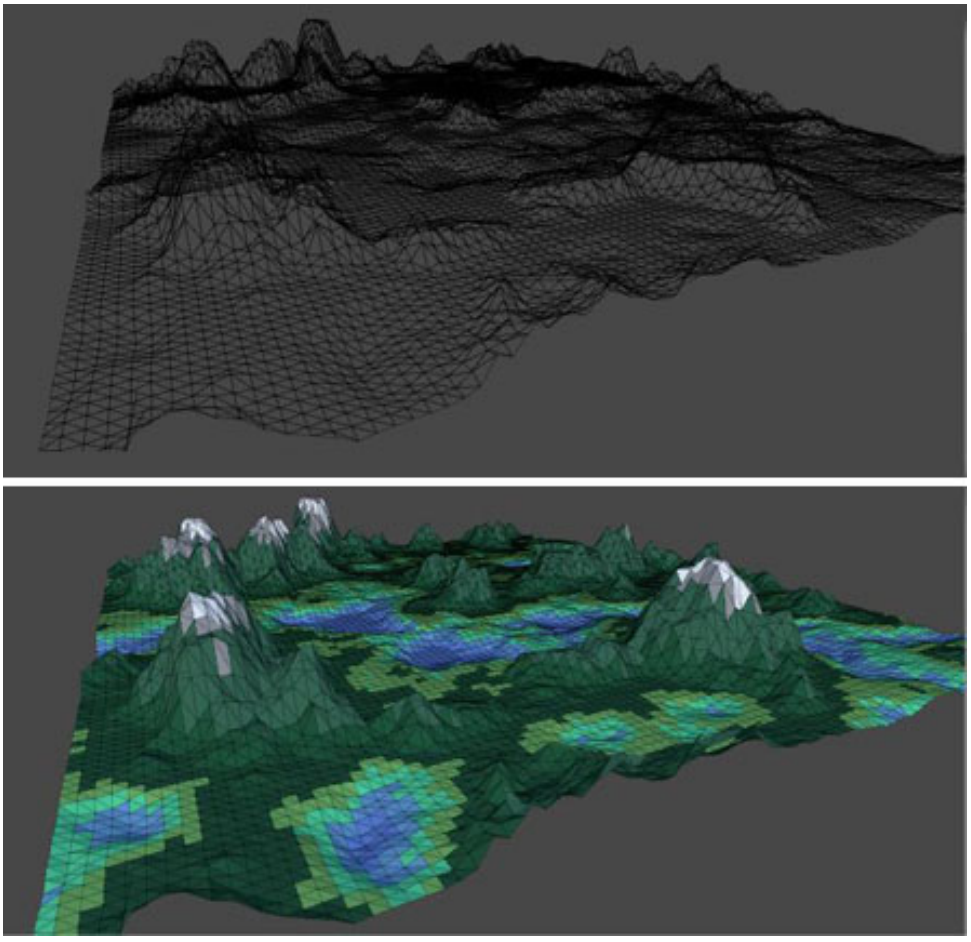


FIGURE 3.14. At the top the 3D mesh generated with the algorithm described in this section. At bottom the same mesh with the texture created with ranges of values of the Section 3.4.3.1.

3.4.3.3 Level of detail

The level of detail, whose acronym is LOD, consist of maintain several models of the same object with a different number of vertices so that between levels of the LOD change the complexity of the polygonal model (that is, change the number of vertices). This technique allows to increase the efficiency since the amount of vertices that have to be processed is decreased. The LOD is used on objects that can be both near or far, or on objects that are very small in the final rendering. Thanks to the LOD, an increase in efficiency can be achieved without sacrificing the final visual result (see Figure 3.15).

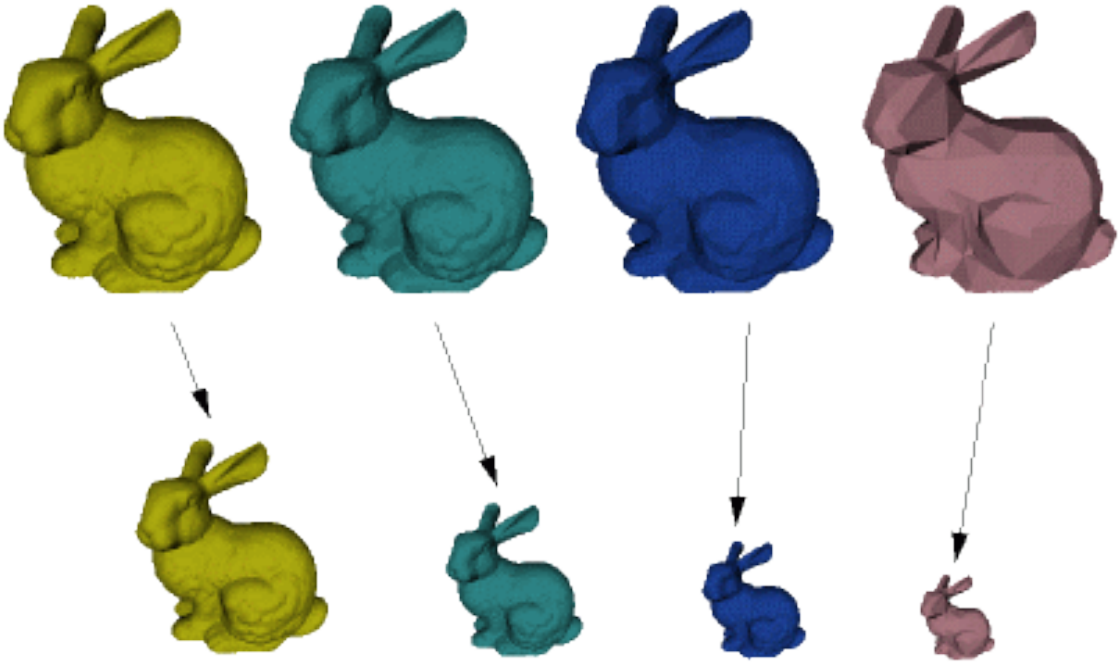


FIGURE 3.15. Example of level of detail [8]. From left to right the level of detail is decreasing.

When generating the mesh, the algorithm of the previous section runs from zero to the width and height of the mesh creating vertices in each point. It can be said that it is increasing by one (i = 1). If a mesh with a width equal to five is taken as an example, raising one by one would generate the five vertices (see Figure 3.16).

FIGURE 3.16. Example of LOD with increment equal to one (i = 1).

If now this increment is equal to two (i = 2), intermediate points will be omitted, and the vertices will be created at points zero, two and four (see Figure 3.17). However, any increment cannot be used. For example, if an increment equal to three is used (i = 3), vertices in zero and three would be created, and then a vertex in point six would be created, but this can not happen since that point does not exist, leaving the mesh incomplete. The increment in the LOD corresponds to a factor of the width minus one. In the previous example, the increment will be: i = !(w-1) = !4 = 1, 2 y 4.

FIGURE 3.17. Example of LOD with increment equal to two (i = 2).

As the Figure 3.18 shows, the value of the LOD is provided to the algorithm, and it is used to traverse the points (with the two nested for's). Also a variable with the number of vertices per line (vPerLine) is necessary to add the triangles to the edge list. The number of vPerLine depends directly on the LOD of the mesh. In the example with five points if the increment is equal to one, the value of vPerLine will be five, but if the increase is equal to two, the value will decrease to three. The formula used to obtain this value can be seen in Figure 3.18.

```
public mesh CreateMesh(heightMap, lodIncrement) {
  mesh = new mesh with width and height equal to width and height of heightMap;
  size of vertexList = w * h;
  size of edgeList = (w-1) * (h-1) * 6;
  index = 0;

  vPerLine = ((w - 1) / lodIncrement) + 1;

  for(y from 0 to h with an increase equal to lodIncrement) {
    for (x from 0 to w with an increase equal to lodIncrement) {
      vertexList[index] = new point(x, curve.evaluate(heightMap[x,y])*multiplier, y);

      if(x < w - 1 and y < h - 1) {
        edgeList.AddTriangle(index, index + vPerLine + 1, index + vPerLine);
        edgeList.AddTriangle(index + vPerLine + 1, index, index + 1);
      }
      index++;
    }
  }
  return mesh;
}
```

FIGURE 3.18. Algorithm (First part) to generate a 3D mesh with LOD.

The Figure 3.19 shows the result of implement LOD in Unity. From top to bottom and from left to right, the level of detail decreases, so the increment increases. From the highest level of detail to the smallest level of detail, the number of vertices changes from fifty-five thousand vertices to one thousand five hundred vertices. Also, the result of this section can be seen in a video by clicking here.

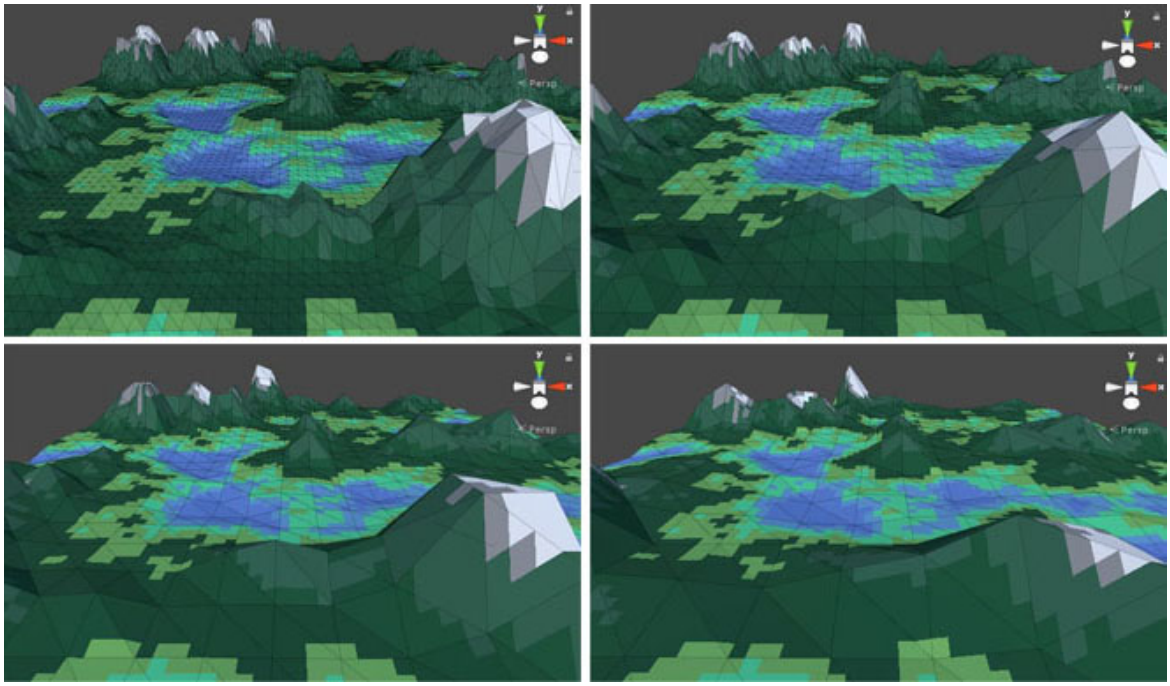


FIGURE 3.19. Result of implement LOD in Unity.

3.4.3.4 Chunks

With the algorithms of sections 3.4.3.1 and 3.4.3.2 and with a high level of detail, a terrain could be already created, but it only could be used for a non-real time simulation, since many vertices would form the ground. It would not be suitable if it is used in a video game because millions of calculations would have to be done in each frame only to render the terrain. Taking into account that a minimum refresh rate equal to 30 fps is expected in a video game, so many calculations are non-viable.

For this reason, some mechanism is needed to maintain that frame rate with a large terrain. One of these mechanisms is the level of detail (see Section 3.4.3.3). In this section another mechanism that will be used is explained and how to combine both to achieve a good result.

The basic idea is to generate only the parts of the terrain, or chunks, that are visible from the player's position. And finally, the level of detail is varied depending on the distance between that chunk and the player. A close chunk will be seen in great detail, and a far chunk will be seen in little detail.

```
public class Chunk {
  chunkPosition; chunkCoord; chunkSize; meshData;
  chunkMultiplier and chunkCurve = same value for all chunks;

  public void Chunk(coord, size){
    chunkPosition = coord * size;
    chunkCoord = coord;
    chunkSize = size;
  }

  public void doInvisble(){
    Make meshData invisible;
  }

  public void doVisible(int lod){
    meshData =  MeshData.CreateMesh(size, chunkMultiplier, chunkCurve, lod);
    Make meshData visible;
  }
}
```

FIGURE 3.20. Pseudocode of Chunk class.

The Figure 3.21 shows how to make visible the chunks that are around the player. To carry out the calculations is needed: player's position, size of the chunks, number of chunks visible from the player's view, a reference to a dictionary where the value is a Chunk object and the key is his position and a reference to a list of Chunks objects that are visible after the last update of the game.

The first step of this algorithm is to hide all visible chunks of the scene. For this, the list that is provided to the method merely is gone through. This list needs to be emptied to complete this step because the chunks visible after the update may be different.

The second step is to obtain the new visible Chunks that are around the player's position. For this, the chunk coordinate where the player is located needs to be known. This coordinate can be known thanks to the player's position and the size of the chunk. For example, if the size of the chunk is equal to one hundred and the player's position is equal to (0, 0) or equal to (50, 20), the coordinate of the chunk where the player is located is equal to (0, 0). However, if the player's position is equal to (120, 20), the coordinate of the chunk is equal to (1, 0).

Once the coordinate of the chunk has been obtained, the algorithm is accessing all the surrounding chunks through its position. If the chunk that is being evaluated has not been created yet, a new object of the Chunk class is created and is added to the dictionary with key equal to its coordinate and value equal to the object itself. Finally, it becomes visible with the right level of detail and is added to the list of visible Chunks objects.

```
void UpdateVisibleChunks(playerPosition, chunkSize, chunksVisible, chunkDictionary,
                         listChunkVisiblesInLastUpdate) {
  foreach(chunk in listChunkVisiblesInLastUpdate) {
        chunk.doInvisible();
  }
  Clear listChunkVisiblesInLastUpdate;

  currentChunkCoord = ((int)playerPosition.x/chunkSize, (int)playerPosition.y/chunkSize)

  for(y from -chunksVisible to chunksVisible) {
    for(x from -chunksVisible to chunksVisible) {
      viewedChunkCoord = (currentChunkCoord.x + x, currentChunkCoord.y + y);

      if(chunkDictionary contain the key viewedChunkCoord) {
        chunk = extract the chunk with key viewedChunkCoord from chunkDictionary;
      } else {
        chunk = new Chunk object with parameters viewedChunkCoord and chunkSize;
        chunkDictionary.Add(key = viewedChunkCoord and value = chunk);
      }

      LODIncrease = DistBetween(viewedChunkCoord and currentChunkCoord)
      chunk.doVisible(LODIncrease);

      listChunkVisiblesInLastUpdate.Add(chunk);
    }
  }
}
```

FIGURE 3.21. Pseudocode of terrain formation with chunks.

A problem that is found when generating chunks with different level of detail is that there will be parts of the two adjacent mesh that will not coincide due to the difference in the number of vertices between them (see Figure 3.22). Nonetheless, it is a negligible problem for the player since it occurs in distant chunks. The closest chunks to the player are always displayed at the highest level of detail.

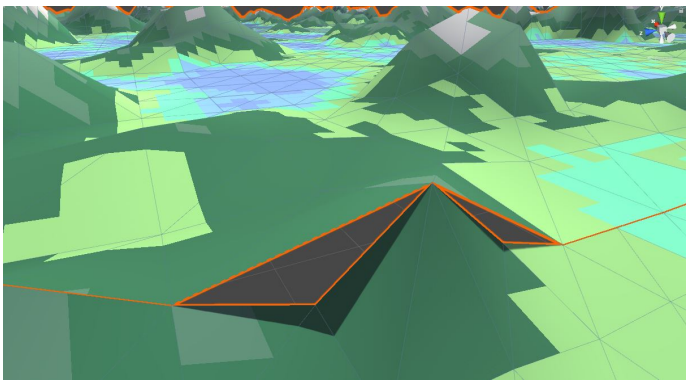


FIGURE 3.22. Difference of adjacent meshes with different LOD.

The result of implementing these two mechanisms together in Unity can be seen in Figure 3.23. Also, it can be seen in a video by clicking here.

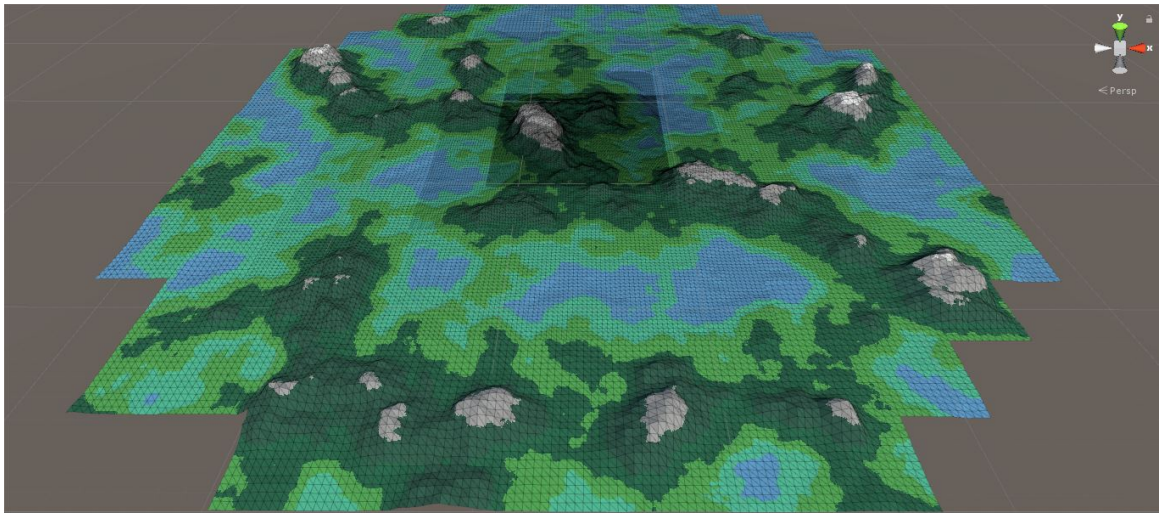


FIGURE 3.23. Result of implementing the chunks with LOD.

3.4.3.5 Threads

Although the terrain is now formed by chunks and in different levels of detail, there may still be considerable drops of fps or freeze of the game. It is due to the high number of calculations that are still required to show the chunks. Depending on the number of chunks that are shown and the level of detail of these, the number of vertices is between five hundred thousand and one or two million approximately. For this reason, it has been decided to implement threads.

A thread allows separating code in multiple tasks. So, dividing the calculations of execution and increasing the efficiency of the program is possible [13]. Resorting to the Thread class of C# [3] is possible to achieve this speed increase when the data of each chunk are calculated. However, a little care must be taken when using the threads within Unity since this class does not belong to him. Because of this, tasks such as access to components of Unity objects cannot be done outside the main Unity thread.

The Figure 3.24 shows a visual scheme where how the flow of actions inside and outside the main Unity thread has been implemented. Two queues have been used: One of them stores the height and color map data and the other one stores the mesh data (vertices and edge). These data are requested from main Unity thread, and later they are calculated and stored outside the main Unity thread. Finally, all stored data is displayed again inside the main Unity thread.

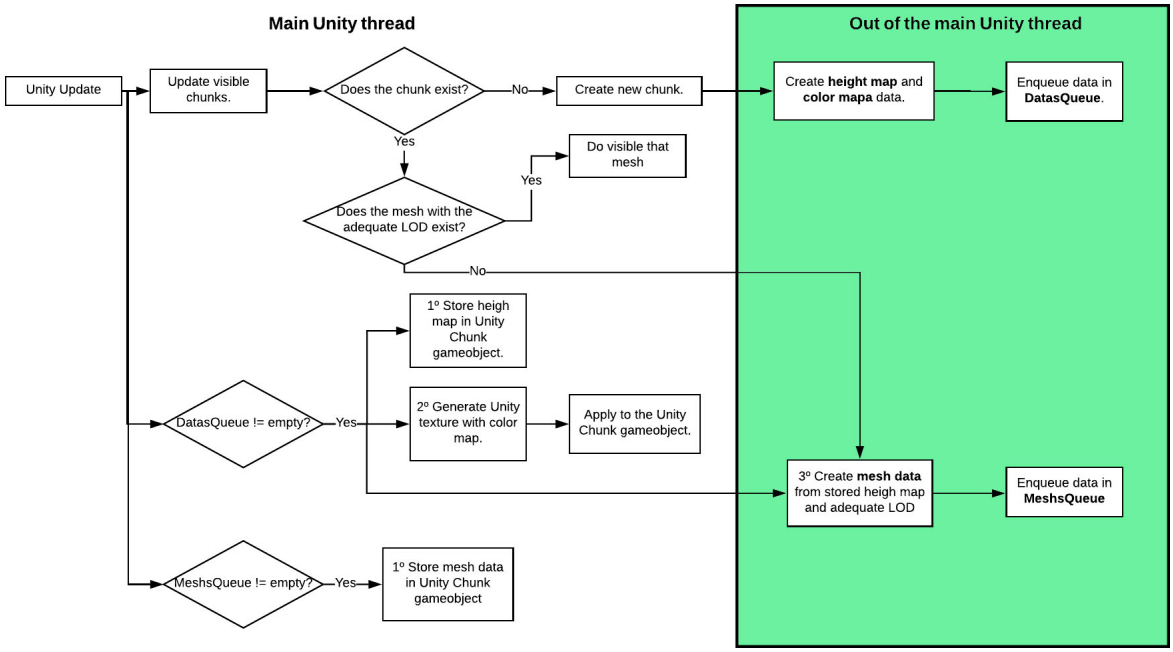


FIGURE 3.24. Flow diagram of the implementation of the threads.

3.4.3.6 Falloff map

Until now, an infinite sequence of chunks is generated procedurally. This result is not suitable for this project since an island-shaped terrain is needed. That is, it has to be surrounded by water. For this, a falloff map will be used. A falloff map is a 2D texture with a gradient. In this case, it is a square gradient from black to white; Or in other words, it is a 2D map with values from 1 to 0 (see Figure 3.25).

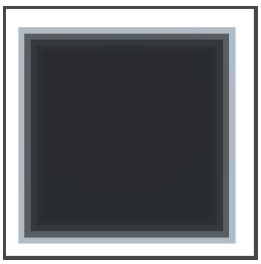


FIGURE 3.25. Falloff map of 40 x 40 px.

Like it happened with the noise map, any bitmap stores numeric values that can be used for different applications. Noise map stores heights. In this case, the falloff map will be used to decrease the height of the vertices that correspond to the periphery of the Falloff map.

In the Figure 3.26 the pseudocode of falloff map generator is displayed. It is worth mentioning the use of a sigmoid function to control where the gradient is produced and its size (see Figure 3.27).

```
float[,] createFalloffMap(int size) {
 map = float array size x size;
  for (i from 0 to size) {
    for (j from 0 to size) {
      x = i / size * 2 - 1;
      y = j / size * 2 - 1;
      value = Maximum(Mathf.Abs(x), Mathf.Abs(y));

      map[i, j] = Use Sigmoid function with x = value, a = 5 and b = 5.2;
    }
  }
return map;
}
```

FIGURE 3.26. Pseudocode of Falloff map generator.

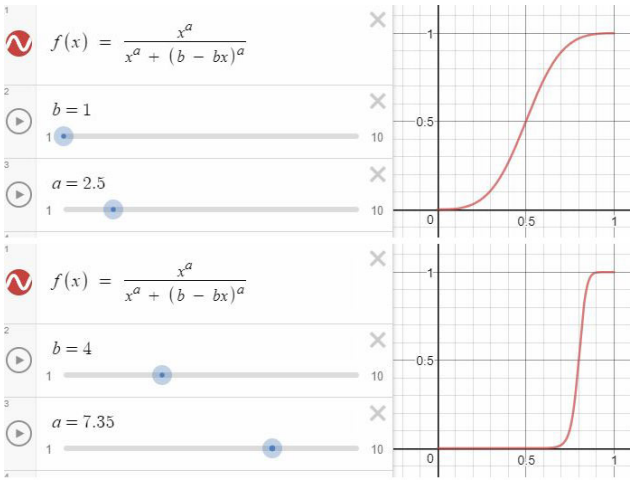


FIGURE 3.27. Sigmoid functions with different values obtained from the Desmos website [5].

Each value of the falloff map is applied to all the vertices of a chunk. So when the noise map of a specific chunk is obtained, the corresponding value of the falloff map is searched and all the values of the noise map are modified.

The result begins to be remarkable, but it presents a problem: the height between two adjacent vertices of different chunks are too different. It is because the values between two contiguous

pixels of the falloff map are different. This problem can be seen in the Figure 3.28.

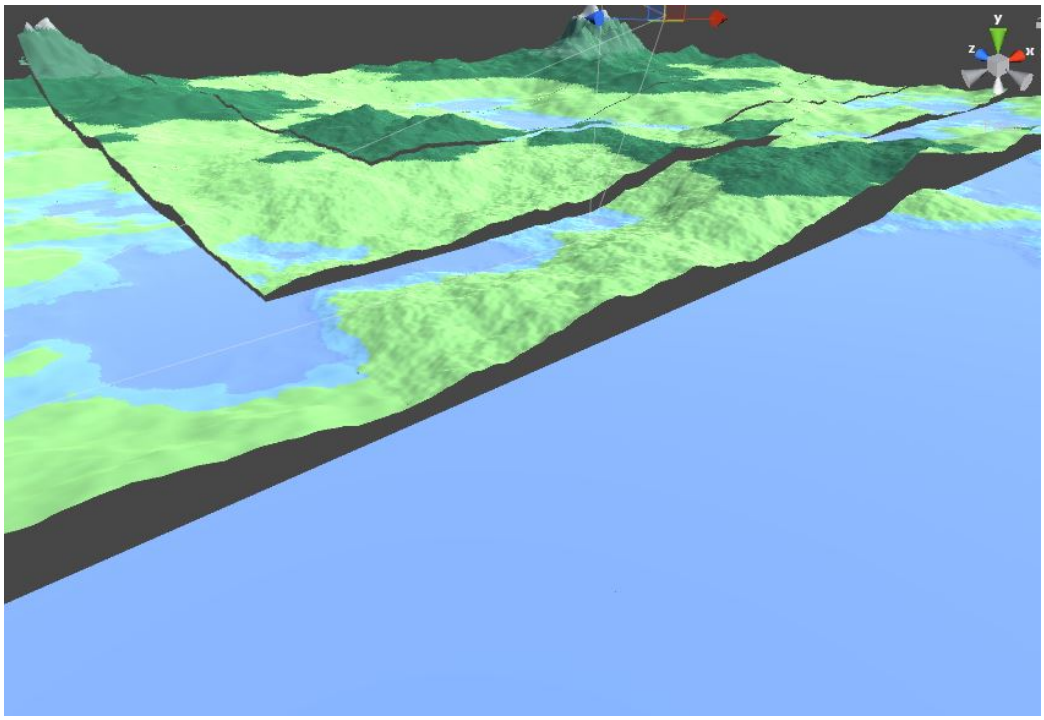


FIGURE 3.28. Problem when the falloff map is directly applied in each chunk.

One might think that this problem is solved by working directly at the vertex level instead of at the chunk level. But for obvious reasons if such a large map were generated, the calculations to obtain it would instantly freeze the system for a few minutes and take up a lot of memory.

To solve this, four adjacent pixels of the Falloff map are taken to determine the final value of each vertex in a chunk:

1. Each value of the four pixels of the falloff map corresponds to one of the four corners of the height map.

2. Bilinear interpolation is done with the four values, and a gradient is obtained (see Figure 3.29).

Gradient map values are those that are subtracted to each value of the height map of a chunk. The pseudocode of the final algorithm can be seen in the Figure 3.30. And the final result obtained after applying the said algorithm to the terrain generator can be seen in the Figure 3.31. Also, it can be seen in a video by clicking here.

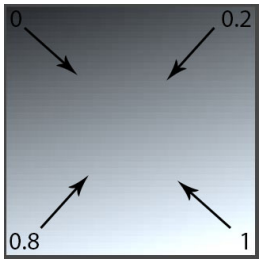


FIGURE 3.29. Example of a chunk gradient for the falloff map.

```
public float[,] applyFalloffMap(falloffSize, heightMap, coordChunk) {
  falloffMap = createFalloffMap(falloffSize);

  falloffMapValueSupLeft = falloffMap[coordChunk.x, coordChunk.y + 1];
  falloffMapValueSupRight = falloffMap[coordChunk.x + 1, coordChunk.y + 1];
  falloffMapValueInfLeft = falloffMap[coordChunk.x, coordChunk.y];
  falloffMapValueInfRight = falloffMap[coordChunk.x + 1, coordChunk.y];

  InterpolationMap = new float array with same dimensions as heightMap;
  InterpolationMap = Generate bilinear interpolation map with the four previous values

  for(y from 0 to width of heightMap) {
    for (x form 0 to height of heightMap) {
      heightMap[x, y] = heightMap[x, y] - InterpolationMap[x, y];
    }
  }
}
```

FIGURE 3.30. Pseudocode of modification of the height map with the falloff and gradient map.

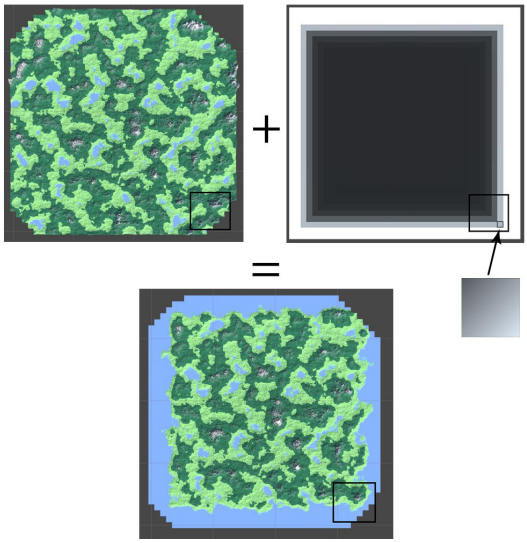


FIGURE 3.31. Final result after applying the falloff and gradient map in Unity.

3.4.3.7 Water

Water was added to the game by using the scripts created by Blendcraft Solutions [24]. This script works in a similar way that the terrain generator seen in the previous sections: First, it generates a plane with the desired dimensions and, subsequently, it obtains the height of each of the vertices using the Perlin Noise function.

3.5 Task 6: Generation of resources

This is the last step to finish the generation of the terrain. Resources are those objects with which the player can interact. As already explained in the GDD (Chapter 2), each geographical area will have its resources. For example, Iron mines are generated only in the Half mountain and Beach biomes.

For the resource generation, an algorithm based on Raycast has been developed, that is, an algorithm composed of different methods has been written where the unity class called Raycast is used. A Raycast consists of throwing a ray from a source position in a specific direction. It returns the first object with which it collides in the scene [22]. It also provides many other values such as the distance between the source position and the object or the global point of the scene where it has collided.

```
void AddPositionResources(chunkPosition, chunkHeightMap) {
  for (i from 0 to numPositionsOfResourcesPerChunk) {
    raycastPosition = chunkPosition + random between -chunkWidth and chunkWidth;
    raycastDirection = - vector Up;
    hit = Raycast(raycastPosition, raycastDirection);

    if (hit != null) {
      height = chunkHeightMap[hit.textCoord.x*chunkWidth, hit.textCoord.y*chunkWidth];
      resourcePosition = collision point of hit;

      int index = 0;
      while (index less than length of biomesList) {
        if (height greater than height of biomesList[index]) { index++;}
        else{ break;}
      }
      finalBiome = biomesList[index - 1];

      Create resource in biome = finalBiome in the position = resourcePosition
    }
  }
}
```

FIGURE 3.32. Pseudocode to determine the biome where to create the resource.

The basic idea of this algorithm is: launches Raycasts randomly from an elevated position in the negative direction of the Y-axis, that is, perpendicular to the XZ plane in the negative direction; Get the position where the ray has collided; And access the Height map to check which biome is in that position.

In the Figure 3.32 the pseudocode to determine the biome is displayed. When this code is executed, an approximate number of resources is created in each chunk (in the pseudocode of the Figure, approximately "numPositionsOfResourcesPerChunk" resources will be generated). So this algorithm will need to be executed only once for each new chunk. The algorithm creates an approximate amount of resources per chunk cause in each position where there will be a resource, a set of these will be generated. This topic will be discussed later.

Following the algorithm, the origin point of the ray is placed in a random position of the chunk to obtain the object with which it collides. This object will always be the terrain. However, the Raycast not only allows access to the object. One of the values what it provides is the texture coordinate of the object where the ray collides. This value is used in this algorithm to obtain precisely the height within the height map (in a range between zero and one -see Section 3.4.3.1-). Once this height is obtained, the list of biomes (ranges of values) is travelled to get the biome to which the collision point of the ray belongs. Finally, the necessary resource is created.

The Figure 3.33 shows the algorithm to generate the resources in the Beach biome as an example. When previously there has been talking about an approximate number of resources per chunk was cause this algorithm works with percentages. If the previous algorithm has determined that resources have to be generated in the Beach biome, there is a seventy percent probability that none will be created, a twenty percent probability that an iron mine will be built, and only a ten percent of the generation of rocks. This algorithm model has been followed to all the terrain biomes by changing and adjusting the percentages for a correct result.

```
AddResourceInSandBiome(resourcePosition) {
  randPercent = Random between 0 and 100;

  if(randPercent less than 20) {
    instanciate Iron mine object in resourcePosition;
    for (i from 0 to random between 2 and 5) {
      instanciate Stone arround resourcePosition;
    }
  }else if(randPercent less than 30) { instanciate Rock object in resourcePosition;
  }else{ return;}
}
```

FIGURE 3.33. Pseudocode to generate resources in the beach biome.

The result after implementing this algorithm to the procedural generation can be seen in the Figure 3.34. Also, it can be seen in a video by clicking here.

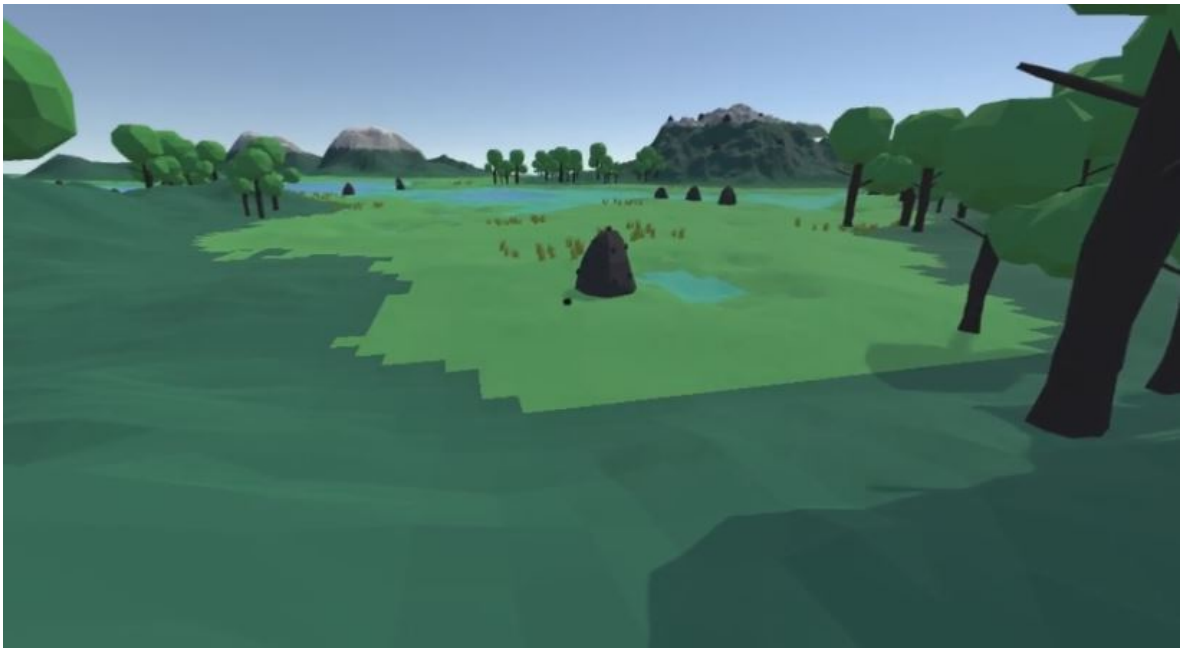


FIGURE 3.34. The final result of generating resources in the game.

3.6 Task 7: Programming of interaction with the enviroment, collection resources and crafting mechanics

For the interaction of the player with the terrain and resources, a small script based on Raycast has been programmed to detect the object with which the player wants to interact. The collected resources are automatically stored in one of the four gaps in the inventory. If there is no gap in the inventory, the player will have to throw any resource he has stored. It is achieved by clicking click on that resource (in one of the four gaps in the inventory) and drag it out of the inventory. This will make it reappear in the terrain.

For the mechanics of crafting, an XML file has been written where all recipes of the resources are saved. Each recipe includes the following attributes: id of the resource, name, number of that type of resources that the recipe gives, whether the resource is or is not a structure and the opening mode to know if the inventory has to be opened in table mode, in furnace mode, etc. Likewise, each recipe contains a list of resources you need to create it. Each resource stores the attributes: id, name and number of resources required by the recipe. Finally, a method that reads

this file and stores a list of XML recipes has been created. For the recipe shown in the Figure 3.35, eight iron plates y five coal ore are needed to craft steel.

```
<RecipeCollection>
  <Recipes>
    <Recipe id="6" name="Steel" number="1" isStructure="0" openingMode="0">
      <Resource id="4" name="IronPlate" number="8"/>
      <Resource id="2" name="Coal" number="5"/>
    </Recipe>
  </Recipes>
</RecipeCollection>
```

FIGURE 3.35. Recipe of steel written in XML.

3.7 Task 8: Memory

This section is about how the memory has been carried out. Like Task 0 (see Section 3.1), the memory of the End-of-Degree Project has been written with Overleaf. Another tool used for this memory has been Tables Generator [20]. This generator has been a great help to add the tables that can be seen throughout the document.

As it has been observed, it has been divided into five chapters: Technical proposal, Game Design Document (GDD), Project development, Results, and Conclusion. It should be noted that all the images that appear in the Chapter 3 have been generated exclusively for this work, except Figure 3.15 of the level of detail section (Section 3.4.3.3). Also, the figures that contain diagrams and sketches of the GDD (see Chapter 2) have been created for this project.

3.8 Task 9: Final presentation

For the final presentation in front of the court, a general review has been prepared for the whole project. Beginning by describing the game that has been created and passing by all the processes that have been carried out for the realization of work: objectives, algorithms of generation, Flow diagrams, etc.

Also for the presentation, a sample video has been prepared where the game is shown running.

Chapter 4

Results

This chapter presents the results obtained after finishing the End-of-Degree project at the Degree of Design and Development of Videogames of the Jaume I University. Also, an analysis of the runtime of the most important algorithms is shown.

4.1 Results

The resulting product is a basic survival game that serves to show the main objective of this project: the procedural generation of optimized maps for survival video games. The terrain of the video game has the shape of an island, and it is generated procedurally and efficiently thanks to the Perlin Noise function.

The following link allows to access to a Google Drive folder where some videos can be found inside a folder called "Videos". Those videos show the running game, how the terrain is generated and the trailer of the game. This trailer can be seen by clicking here.

Google Drive folder

The written codes in C# language that have been created for this project can be seen by clicking here.

Clicking here gives access to another GitHub repository where the Unity project of the game is located. Also, the executable game is inside the same repository. It is necessary to download the entire folder to play Alone since there are files that Unity does not compile and they are necessary to it.

4.2 Project in numbers

For the generation of the terrain, nine classes have been written in C# language. Some of them simply serve to store data within the same context (for example, the Biomes class only stores the name of the biome, height where it is found and color to represent it in the texture of the terrain). For the game, thirteen classes have been written in C# language and a script in XML language. In the Table 4.1 a summary of all the written classes is shown.

Terrain generator classes		
Name	Number of lines	Number of methods
Biomes	10	0
Chunk	240	8
DataOfResources	70	5
ProceduralGenerator	1160	42
InfoLOD	55	3
DataOfMesh	95	6
MeshWithLOD	90	6
HeightMapAndColorArray	35	3
ThreadInfo	40	2
Total	1795	75
Game classes		
Name	Number of lines	Number of methods
ConstantAssistant	60	0
ResourceXML	10	8
Recipe	20	0
RecipeContainer	20	2
Recipes (XML)	65	0
BoxColliderController	25	2
NaturalResources	140	8
NoNaturalResource	100	6
Inventory	290	12
Player	75	5
ButtonInventoryHUD	265	6
ButtonInventoryCanvas	150	4
DropdownController	110	5
InventoryCanvasController	40	3
Total	1370	61

TABLE 4.1. Summary of the classes for the project.

Regarding the resources of the game, twenty-six resources have been modeled for this project. All of them can be seen in Appendix B. The individual images are in: Google Drive inside a folder called "Models - Images".

4.3 Analysis

An analysis of the times of each algorithm is carried out in this section with different configurations (see Table 4.2). Each result corresponds to the sum of the time it takes to execute the algorithm each time, that is, if the algorithm has been run three times and each execution has carried out for 30 ms, then the analysis will show 90 ms.

Configurations			
Variables	Test 1	Test 2	Test 3
Chunk with LOD = 1	0 to 200	0 to 550	0 to 100
Chunk with LOD = 3	200 to 550	550 to 650	100 to 200
Chunk with LOD = 4	550 to 650	650 to 750	200 to 300
Chunk size	96	96	96
Terrain size (chunks)	30	50	20
Resources per chunk (Aprox)	20	50	20

TABLE 4.2. Configurations for the analyzes.

The results of the analysis can be seen in the Tables 4.3:

Results			
Algorithm	Test 1	Test 2	Test 3
Generate noise map	2979.341 ms	3075.047 ms	518.564 ms
Apply Falloff map to noise map	638.894 ms	829.548 ms	95.935 ms
Generate mesh	207.123 ms	789.320 ms	91.667 ms
Update visible chunks	678.043 ms	2314.201 ms	566.546 ms
Add resources	175.340 ms	3053.356 ms	60.567 ms
Stats			
PFS	110 - 130 fps	25 - 35 fps	140 - 180 fps
Rendered vertices	2 - 3 M	15 - 18 M	1 - 2.5 M
CPU response time	8 - 10 ms	30 - 45 ms	6 - 7 ms

TABLE 4.3. Results of the analysis.

These analyzes have been done with a computer by the following characteristics:

SO : Windows 8.1 Pro 64-bit

Processor : Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz (8 CPUs)

RAM : 8GM

Graphic card : NVIDIA GeForce GT 740

DirectX Version : DirectX 11

CHAPTER 5

CONCLUSIONS

In this chapter the objectives appointed in the Technical Proposal of this memory are reviewed and commented. In addition, possible future lines of the project and a small personal reflection are addressed.

5.1 Goals

All the objectives described in the technical proposal (see Chapter 1) have been achieved throughout the development of the project:

Goal 1 "Creation of the base of a video game of survival so that, later, it can continue to improve aspects of the game and adding new ones."

A basic survival video game has been created that it serves as a perfect sample to verify the potential and the results of the procedural generation within it. It does not have much playful content since the development of the project has focused on the creation of the world and not on the content of it. There are also some resources described in the GDD that cannot be obtained within the game although they are modelled (for example tools like axes). Nevertheless, continue with its growth to achieve a more entertaining and long game is possible by adding new resources, structures, tools, etc.

Goal 2 " Achieve a physical and logical realism in the procedurally generated terrain."

Physical and logical realism in the generation of the video game terrain has been achieved very satisfactorily. A different island has been generated procedurally in each execution thanks to a 2D noise function.

Goal 3 "Obtain a logical coherence in the procedural generation of resources by the map."

In the generation of resources, the logic described in the Game Design Document has been obtained, that is, each geographical zone has the characteristic resources that were specified in the GDD. However, this generation has been implemented randomly and not procedurally, that is, if the execution is started without changing the input data, the creation of the resources will be different in each execution. Even so, the result is very consistent with what was expected.

Goal 4 "Get an optimized game that can be played by any player on any standard computer."

Chapter 4 shows an analysis with varied terrain configurations. The results obtained in these tests are satisfactory taking into account the size of the terrain that is generated and also the code allows to easily configure the parameters for different computers.

Goal 5 "End up having an attractive visual section for the public. This includes: models, textures, lighting, animations, etc."

The resources and structures of the game have been modelled and textured. Even so, it is the goal that has been weaker in this project compared to the rest. The visual section of the game has a lot of room for improvement for the future. However, there has been satisfied with it.

5.2 Possible future lines

A series of possible improvements have been detected both in the video game and in the generation of the terrain:

- If increasing the efficiency of the game should be necessary, the algorithm for generating the noise map could be changed to a more efficient one. For example, the Simplex Noise algorithm is more efficient than Perlin Noise.

- Every game needs a saved to be able to return it at another time. It would be a good addition to implement a storage system.

- Another aspect would be the implementation of a window that allows the users to configure some values so that the game can be adapted to their needs. For example, the number of chunks that are rendered or the level of detail of these chunks.

- Regarding the artistic section, from the visual as the sound, is one of the features that have more possibilities for change. Adding custom sounds, improvements in resource textures or new resources would be possible changes.

5.3 Personal reflection

As a final reflection, the experience gained with this work has been very satisfactory at a general level. It has served me to consolidate programming knowledge such as the use of threads within Unity or the creation and modification of meshes within Unity. It has also helped me to understand better the process of procedural creation within video games. I look the games that use this technique, whether to create terrains or dungeons, with a different perspective and I can imagine the process behind them.

APPENDIX A

HOURS DEDICATED

This appendix shows a table (see Table A.1) where the performed tasks and the dedicated hours to each of them are represented. It also shows the order that has been followed. Eventually, A table is shown where is compared the estimated hours and final hours (see Table A.2).

In Table A.1, each column represents the tasks accomplished in the project and each row corresponds to a range of ten hours. So the first task was to write the technical proposal and it took up the first ten hours of the project. Subsequently, the GDD and the modelling of resources were accomplished.

Most of the estimated hours in the initial planning of the technical proposal (see Chapter 1) have been completed approximately, even taking less time than expected. The writing of the memory has been the only one task that has required more time. In the initial planning, fifty hours was estimated and the time invested has been doubled. It has been possible thanks to many tasks have taken fewer hours than expected, or even have not required work time (for example, Task T4). See Table A.2 .

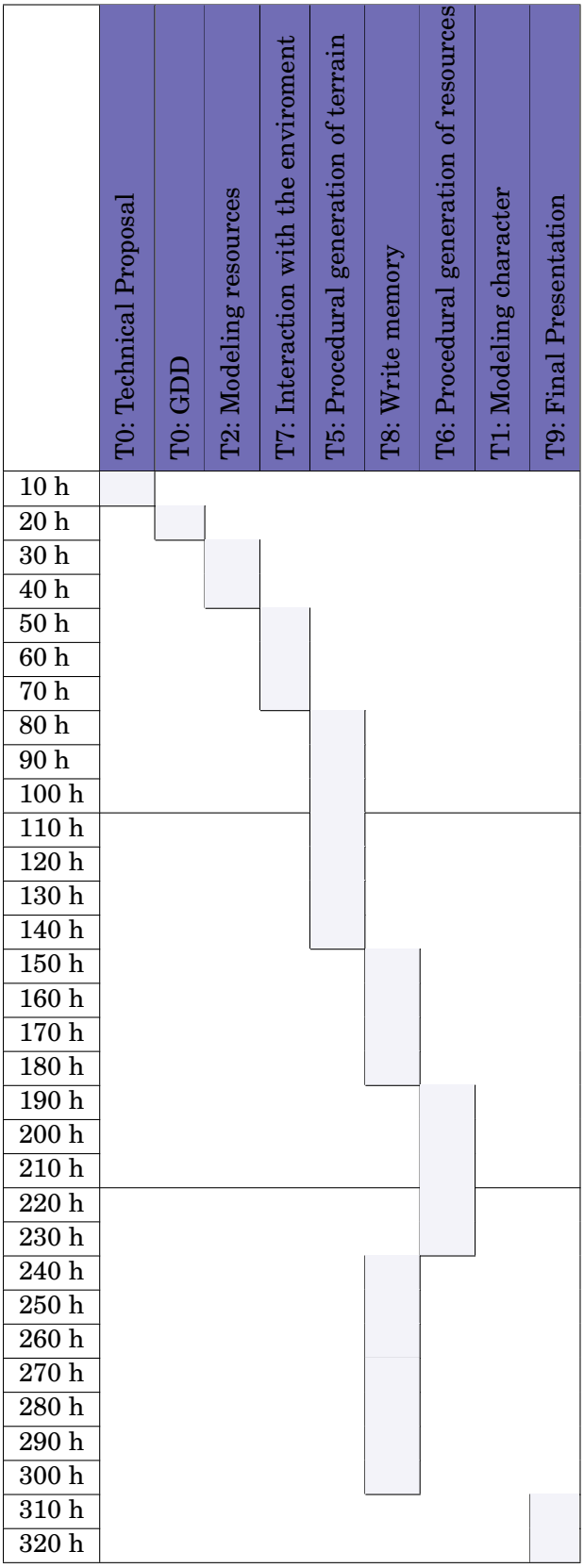


TABLE A.1. Hours dedicated to the project.

Task	Estimated hours	Final hours
T0: Technical Proposal	10	10
T0: GDD	20	10
T1: Modeling character	15	0
T2: Modeling resource	15	20
T3: Modeling game structures	20	0
T4: Main character movement	10	0
T5: Procedural generation of terrain	75	70
T6: Procedural generation of resources	30	50
T7: Interaction with the enviroment	35	30
T8: Write memory	50	110
T9: Final Presentation	20	20
Total:	300	320

TABLE A.2. Comparison of project hours.

APPENDIX B

RESOURCE MODELS

This appendix allows to see the final 3D models of the resources of the game (see Figure B.1). All images can be found in Google Drive inside a folder called "Models - Images"

FIGURE B.1. Resources models.

BIBLIOGRAPHY

[1] Anon, (n.d.). (2018) *Ruido Perlin | Khan Academy* [online] Available at: https://es.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise. [Accessed 7 Apr. 2018].

[2] Astroneer Wiki. (2018). *Astroneer*. [online] Available at: https://astroneer.gamepedia.com/Astroneer [Accessed 19 May 2018].

[3] Bevins, J. (2007). *libnoise: Generating coherent noise*. [online] Libnoise.sourceforge.net. Available at: http://libnoise.sourceforge.net [Accessed 29 Apr. 2018].

[4] Definición de Craftear (2013). *GamerDic, Diccionario online de términos sobre videojuegos y cultura gamer* [online] Available at: http://www.gamerdic.es/termino/craftear [Accessed 25 Feb. 2018].

[5] Desmos.com. (n.d.). *Desmos | Beautiful, Free Math*. [online] Available at: https://www.desmos.com/ [Accessed 24 May 2018].

[6] Docs.unity3d.com. (n.d.). *Unity - Scripting API: Mathf.PerlinNoise*. [online] Available at: https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html [Accessed 3 Apr. 2018].

[7] Foundation, B. (n.d.). *About — blender.org*. [online] blender.org. Available at: https://www.blender.org/about/ [Accessed 14 May 2018].

[8] Garland, M. (n.d.). *3D Computer Graphics :: Levels Of Details*. [online] Mkrus.free.fr. Available at: http://mkrus.free.fr/CG/LODS/ [Accessed 8 Apr. 2018].

[9] GitHub.com (n.dd). *Build software better, together.* [online] Available at: https://github.com/ [Accessed 22 May 2018].

[10] González Hernández, A. (2015). *Generación aleatoria de terrenos 3D con Unity.* [ebook] Available at: https://riull.ull.es/xmlui/bitstream/handle/915/1395/Generacion%20aleatoria%20de%20terrenos%203D%20con%20Unity.pdf?sequence=1 [Accessed 17 May 2018].

[11] IMDb. (n.d.). *TRON (1982).* [online] Available at: https://www.imdb.com/title/tt0084827?ref_=fn_al_tt_2 [Accessed 21 May 2018].

[12] Ironbellystudios.com. (2018). *GDD Template – Ironbelly Studios*. [online] Available at: https://ironbellystudios.com/gdd-template/ [Accessed 15 Feb. 2018].

[13] Jackson, C. (n.d.). *¿Qué son los hilos de un procesador?*. [online] Available at: https://techlandia.com/son-hilos-procesador-info-338189/ [Accessed 17 Apr. 2018].

[14] Lucidchart. (n.d.). *Online Diagram Software & Visual Solution | Lucidchart*. [online] Available at: https://www.lucidchart.com/?noHomepageRedirect=true [Accessed 2 Jun. 2018].

[15] Minecraft.net. (n.d.). *Sitio oficial*. [online] Available at: https://minecraft.net/es-es/ [Accessed 11 Feb. 2018].

[16] No Man's Sky. (n.d.). *Atlas Home*. [online] Available at: https://www.nomanssky.com/ [Accessed 11 Feb. 2018].

[17] Overleaf.com. (n.d.). *About Overleaf.* [online] Available at: https://www.overleaf.com/about [Accessed 14 May 2018].

[18] Overleaf.com. (n.d.). *Overleaf: Real-time Collaborative Writing and Publishing Tools with Integrated PDF Preview.* [online] Available at: www.overleaf.com [Accessed 24 May 2018].

[19] Pedreño Moya, V. (2014). *Generació procedural de mapes per a un JRPG* [ebook] Available at: https://repositori.upf.edu/bitstream/handle/10230/22916/Pedre%C3%B1oMoya_2014.pdf?sequence=1 [Accessed 17 May 2018].

[20] Tablesgenerator.com. (n.d.). *Create LaTeX tables online.* [online] Available at: https://www.tablesgenerator.com/ [Accessed 30 May 2018].

[21] Technologies, U. (n.d.). *Unity - Manual: Packages.* [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/Manual/AssetPackages.html [Accessed 14 May 2018].

[22] Technologies, U. (n.d.). *Unity - Scripting API: Physics.Raycast.* [online] Docs.unity3d.com. Available at: https://docs.unity3d.com/ScriptReference/Physics.Raycast.html [Accessed 11 May 2018].

[23] Unity. (n.d.). *Unity.* [online] Available at: https://unity3d.com/es [Accessed 1 Jun. 2018].

[24] YouTube. (2017). *[Unity Tutorial] How to Make Low Poly Water.* [online] Available at: https://www.youtube.com/watch?v=3MoHJtBnn2U [Accessed 2 Jun. 2018].